Software Improvement Group

On Software Developers Happiness and Code Quality

Yiannis Kanellopoulos

November 11, 2017

Who are we?

In addition to our consultancy work, SIG promotes the creation of maintainable software by teaching and coaching developers, through books, trainings, and evaluating student projects

Your host today

Yiannis Kanellopoulos

- SIG Practice Leader Greece
- Co-Founder Orange Grove Patras

Late ‘90s: Heroes of the *.com era when e-* was dominant

A Message


WIRED
Quick!
The future
is all about>>> people
William Gibson
Reveals His
Secret eBay
Obsession
The Truth
About
Vibrators
Daytrading:
The Net's
Extreme
Sport
Phone Sects:
What Happened
When the
Amish Went
Wireless
Salon's Web
Dream

A confession

Not only the future is all about people,
but most importantly it is (and will be) created by people for people.

The question

Do we care about the happiness of our software developers?

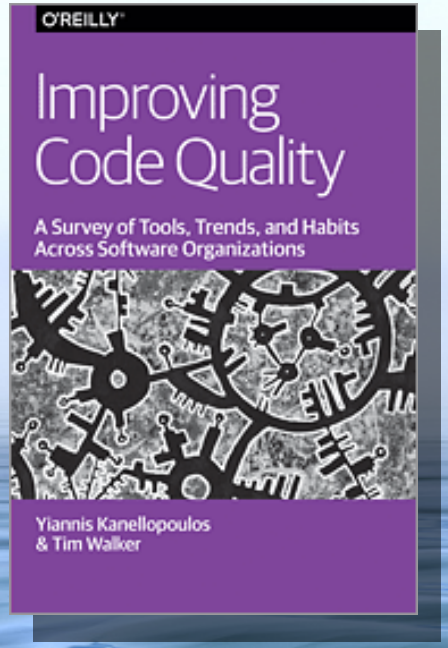
The Academic Evidence

Table 2: Top 10 Causes of Unhappiness, Categories, and Frequency

Cause	Category	Freq.
Being stuck in problem solving	software developer's own being	186
Time pressure	external causes → process	152
Bad code quality and coding practice	external causes → artifact and working with artifact → code and coding	107
Under-performing colleague	external causes → people → colleague	71
Feel inadequate with work	software developer's own being	63
Mundane or repetitive task	external causes → process	60
Unexplained broken code	external causes → artifact and working with artifact → code and coding	57
Bad decision making	external causes → process	42
Imposed limitation on development	external causes → artifact and working with artifact → technical infrastructure	40
Personal issues – not work related	software developer's own being	39

Daniel Graziotin, Fabian Fagerholm, Xiaofeng Wang, and Pekka Abrahamson 2017. On the Unhappiness of Software Developers.
In Proceedings of 21st International Conference on Evaluation and Assessment in Software Engineering, Karlskrona, Sweden, June 15–16 2017 (EASE '17)

The practitioners' viewpoint



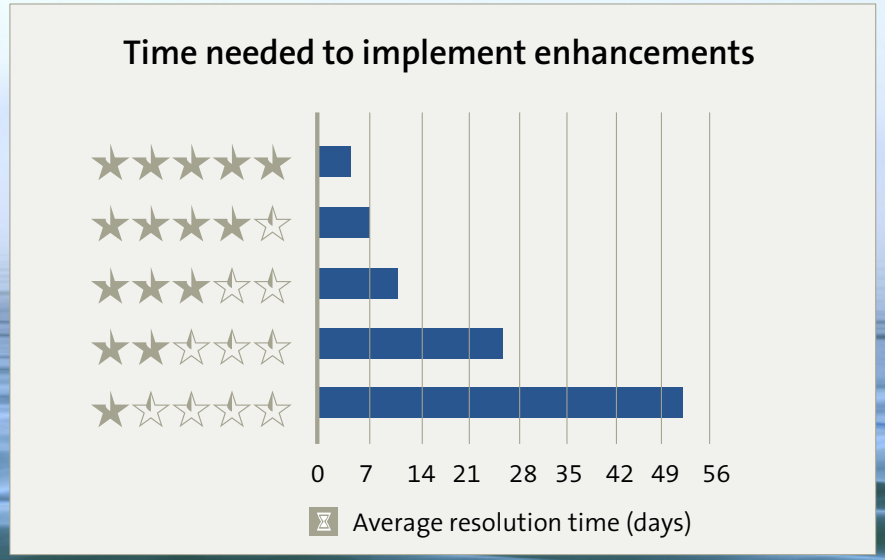
- *Lack of accountability when it comes to code quality,*
- *Limited resources and management support to invest in tools and methodologies,*
- *Lack of awareness when it comes to code quality and modern methodologies.*

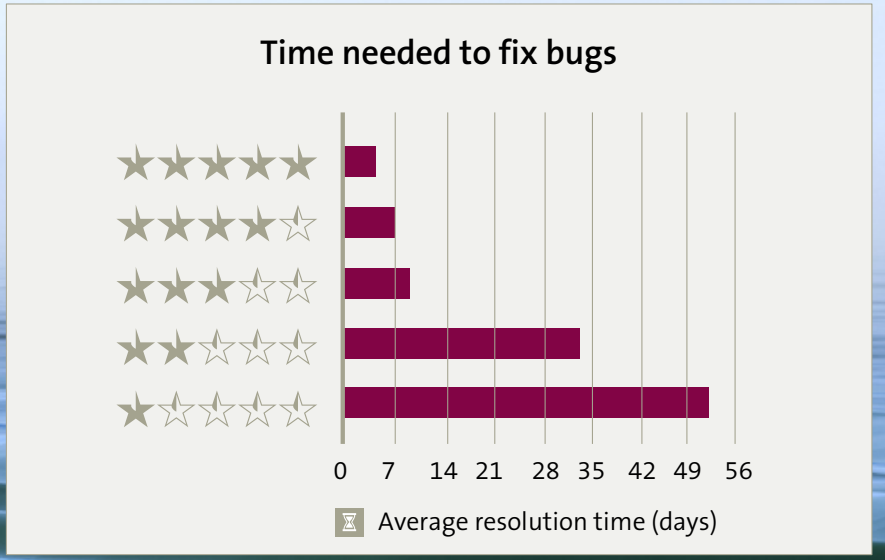
For the managers among us

The business case of code quality

A higher software product quality leads to:

- The faster implementation of improvements and the solution of defects
- The throughput rate improves by factor 3.5 to 4.0 between 2 and 4 stars
- **Plus** higher retention rates and job satisfaction based on the experiences from our own clients.





Source: "Faster issue resolution with higher technical quality of software", Software Quality Journal, 2011

My 0.02$ of wisdom

Software developers need to be **taken care** of
and rest assured that they will **return** the **favor.**

Software Improvement Group

About us

2000– 2005

2000– 2010

Since 2010

SIG has the largest software quality database in the world

235
technologies

2,507
monitored systems

227
clients

50
source code uploads p/week

22,000
inspections

>7,400,000,000
lines of code

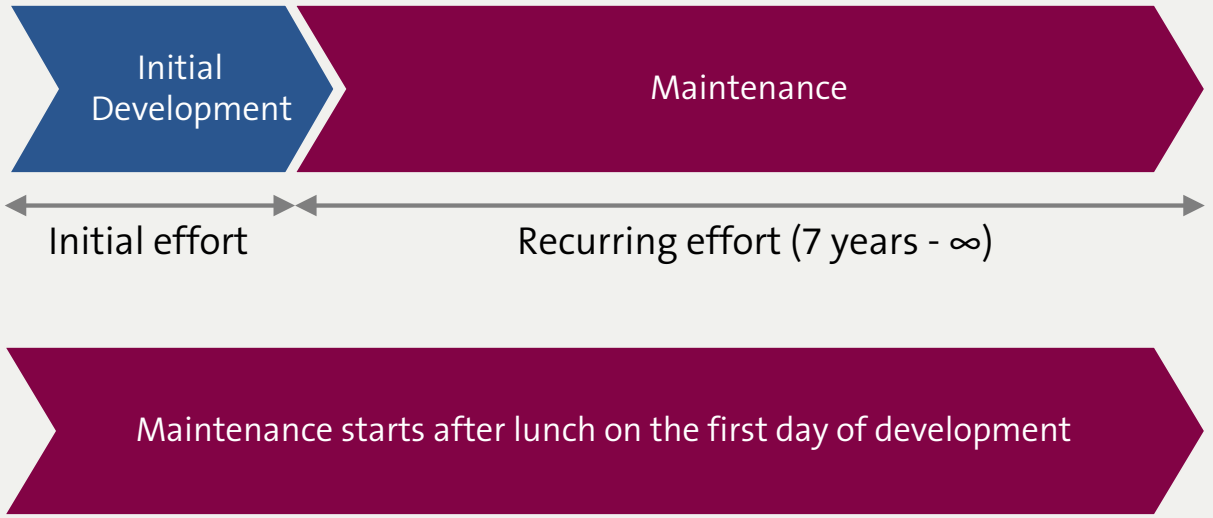
Our perspective on software

" Software is the DNA of today's Information Society",
Prof. Joost Visser, CTO SIG

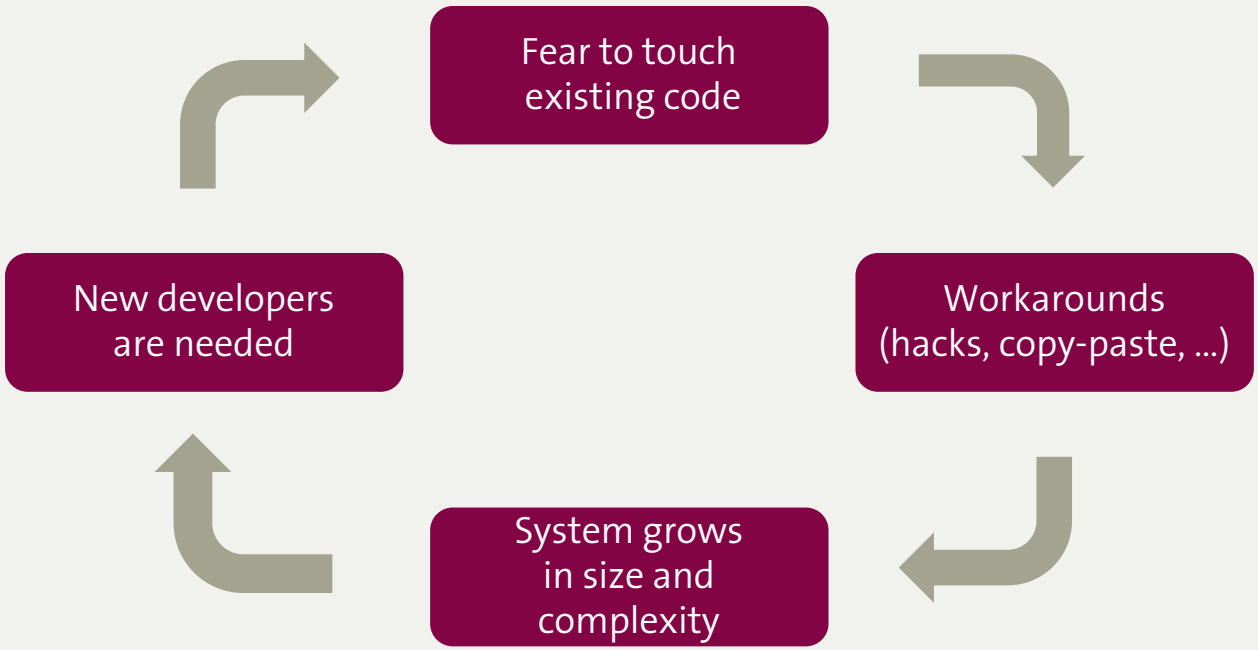
A statement

- You are here thanks to software, but hopefully you did not notice.
- Hopefully the navigation software did not send you into a dead-end street.
- Hopefully your payment went through in a matter of seconds.
- **Good software performs its functions unnoticed.**
- Good software does not get in the way of its users to do what they want to do and
  - in this sense good software remains invisible.
- The moment that software gets noticed is the moment when it fails. When your
  - car does not start and needs to be towed to the garage for a reboot.*

Why maintainable software?



The Vicious Cycle of Unsustainable Software Development



Giving feedback on software products: personal versus tool-based

Personal feedback

- Specific for your project
- More sensitive to context
- Concrete suggestions to improve

Tool-based feedback

- Faster feedback loop
- Allows for scalability by iteration
- More objective

So which one is better?

There is a false dichotomy between full automation and human intervention. Successful quality control combines tool-based measurement with manual review and discussion.

Pitfalls in using measurements

One-track metric

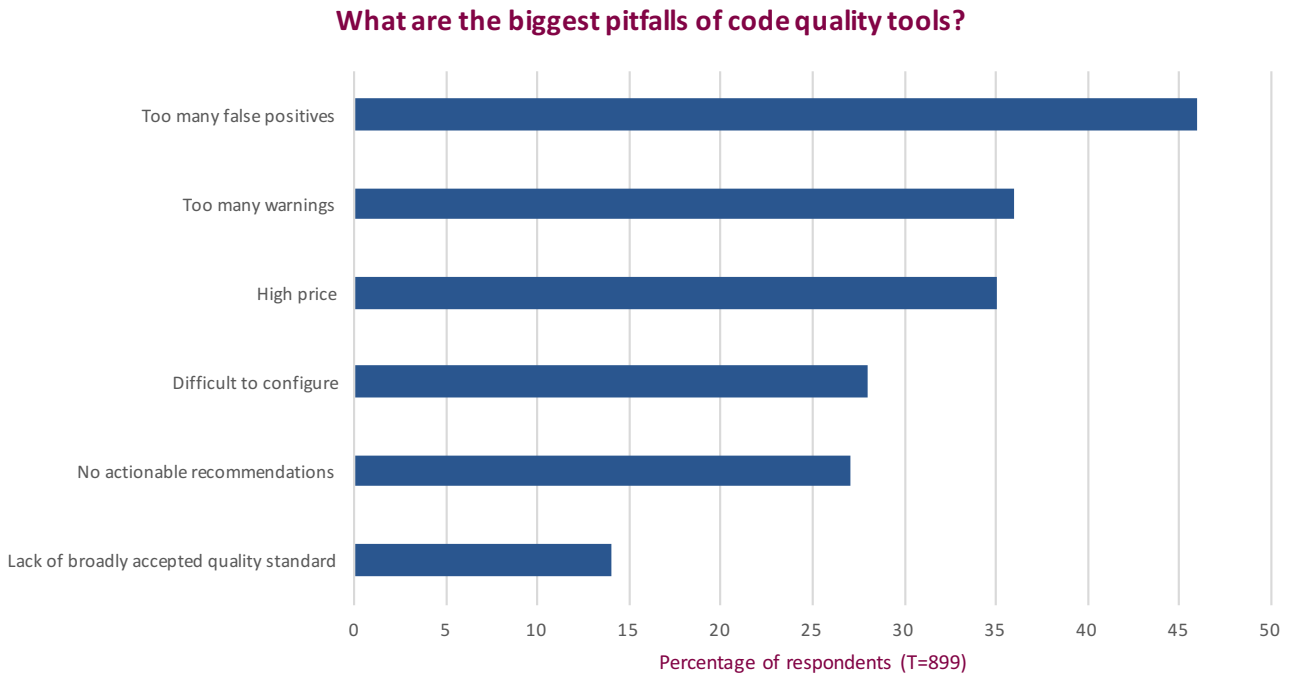
Metrics Galore

Treating the metric

Metric in a bubble

Source: "Getting what you measure", Eric Bouwers

Static code analysis challenges

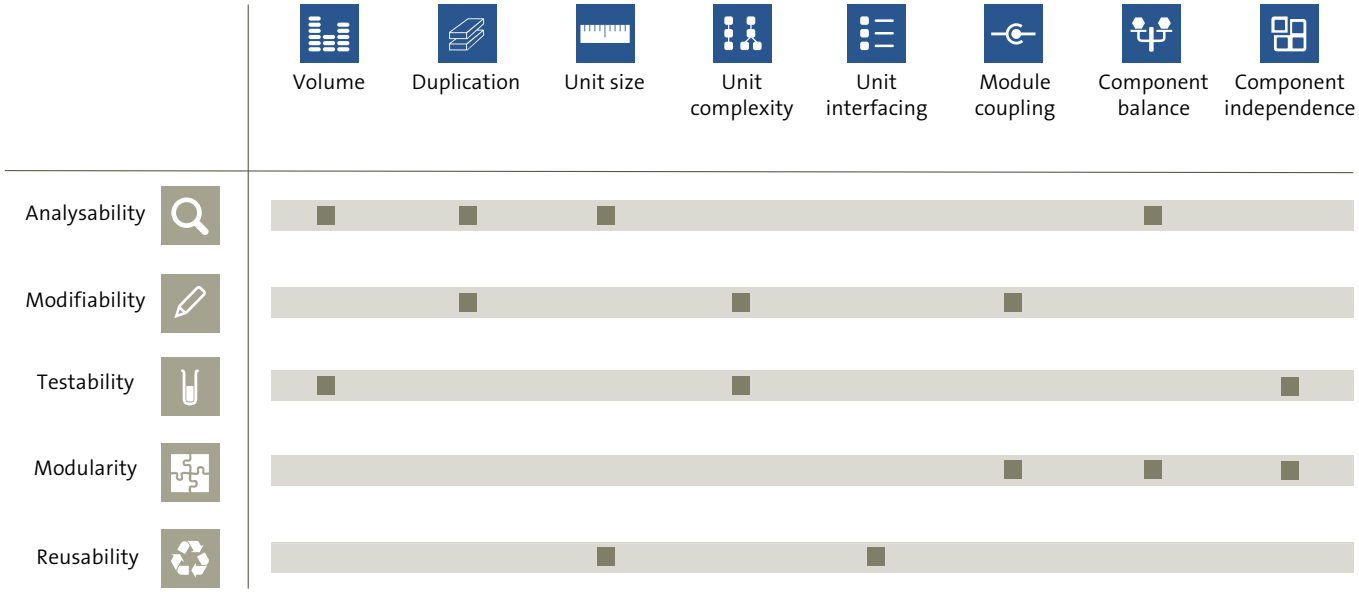


Source: "Improving code quality"

ISO/IEC 25010: Standard for Software Quality

> SIG ISO/IEC 25010 Certified Capabilities

SIG Quality Model Maintainability



The measurements of the implementation lead to a benchmark score (in stars from ★☆☆☆☆ to ★★★★★) where ★★★☆☆ is market average.

Maintainability according to ISO 25010

Sub-characteristics explained

"Degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers."

Sub-characteristic	Definition
Analyzability	Degree to which it is possible to assess the impact of an intended change.
Modifiability	Degree to which the product can be modified without introducing defects or degrading product quality.
Testability	Can test criteria be established for a product and can tests be performed to determine whether those criteria have been met?
Modularity	Is the product composed of components such that a change to one component has minimal impact on other components?
Reusability	Degree to which an asset can be used in more than one system.

Volume

Keep your codebase small

A larger system is less maintainable

- Harder to find the place to make a change
- Harder to test

Measured as the estimated effort in number of man months to rebuild the current code base:

```
/* Add a command to the chain.
 * @param command the new command
 */
public void addCommand(Command command) {
    if(command == null)
        throw new IllegalArgumentException();

    if(_frozen)
        throw new IllegalStateException();

    Command[] results = new Command[_commands.length + 1];
    System.arraycopy(_commands, 0, results, 0, _commands.length);
    results[_commands.length] = command;
    _commands = results;
}
```

10 Lines of Code (LOC) ➔ $\frac{10}{Factor_{Java}}$ Man months

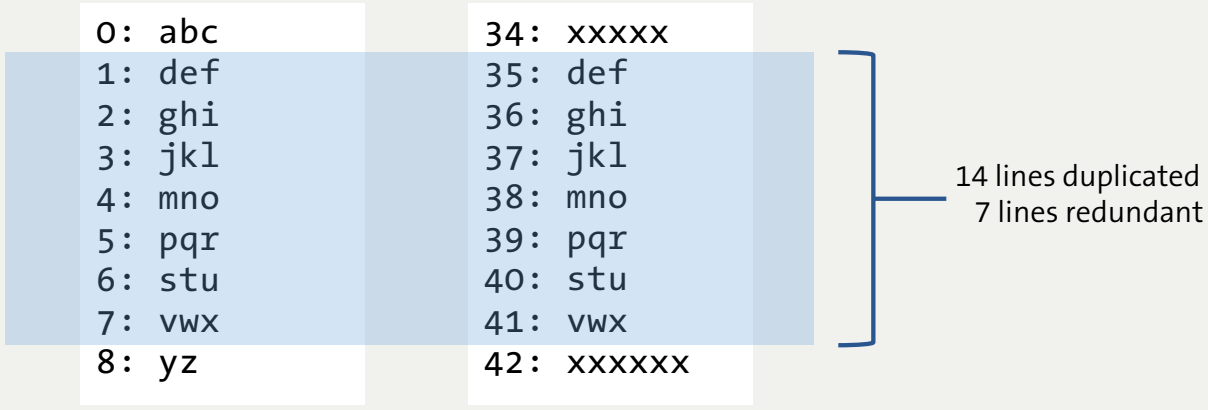
Duplication

Write code once

Duplication of code reduces changeability and analyzability

- Substantial duplication makes bug fixing harder
- Substantial duplication makes testing harder

Measured as the percentage of redundant code:



Unit size

Write short units of code

Small units are easier to reuse, analyze and test than long units

- A unit is a function, method or procedure
- Small units can more easily be reused
- Small units have a lower complexity
- Small units can more easily be documented via their names

Measured by means of a risk profile using unit length in lines of code:

7 LOC in unit

```
Collection<Package> getChangedPackages() {
    if (changedPackages != null) {
        return changedPackages;
    } else {
        return Collections.emptyList();
    }
}
```

Unit length (LoC)	Risk estimation
1-15	Easy to understand, no risk
16-30	Average unit size, low risk
31-60	Long unit, moderate risk
> 60	Very long unit, high risk

Unit complexity

Write simple units of code

Simple units are easy to modify and to test

- A unit is a function, method or procedure
- Units with low complexity are easier to test
- Units with low complexity are easier to modify

Measured by means of a risk profile based on McCabe* complexity of units:

McCabe: 4

```
void removePackage(String name, RuleBase rb) {
    Package[] ps = rb.getPackages();
    if (ps == null) return;
    for (int i = 0; i < ps.length; i++) {
        Package p = ps[i];
        if (p.getName().equals(name)) {
            rb.removePackage(name);
            return;
        }
    }
}
```

Cyclomatic complexity	Risk estimation
1-5	Clear code, low risk
6-10	Complex, moderate risk
11-25	Very complex, high risk
> 25	Not understandable, untestable, very high risk

** after: McCabe, IEEE Transactions on Software Engineering, 1976*

Unit interfacing

Keep unit interfaces small

Unit with small interfaces are easier to understand and reuse

- Unit interfacing can be measured as the number of parameters of a unit
- Units with many parameters do not provide proper encapsulation
- Unit interfacing can be improved by encapsulating multiple parameters in a single object

Measured by means of a risk profile based on the number of parameters of units:

Number of parameters: 2

```
void removePackage(String name, RuleBase rb) {
    Package[] ps = rb.getPackages();
    if (ps == null) return;
    for (int i = 0; i < ps.length; i++) {
        Package p = ps[i];
        if (p.getName().equals(name)) {
            rb.removePackage(name);
            return;
        }
    }
```

Number of parameters	Risk estimation
0-2	No risk
3-4	Low risk
5-6	Moderate risk
>6	High risk

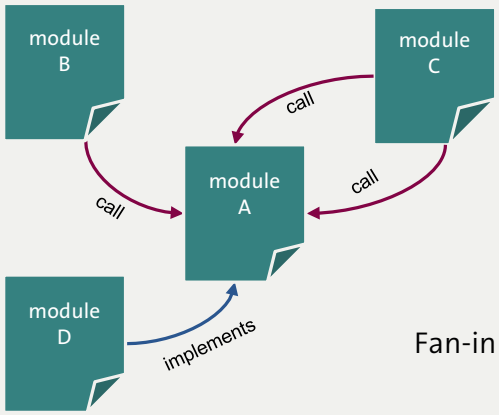
Module coupling

Separate concerns in modules

A loosely coupled code base is easier to change

- Small, loosely coupled modules allow developers to work on isolated parts of the code base
- Small, loosely coupled modules ease navigation through the code base
- Avoid large modules in order to achieve loose coupling between them

Measured by means of a risk profile as the fan-in between modules:



Fan-in: 3

Module coupling (fan in on file level)	Risk estimation
0-10	Easy to stabilize, no risk
11-20	Low risk
21-50	Moderate risk
>50	High risk

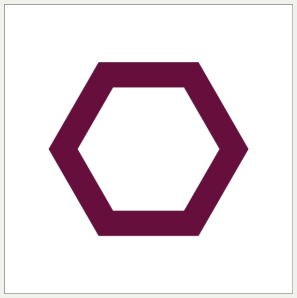
Component balance

Keep architecture components balanced

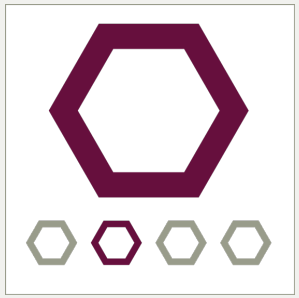
A well-balanced codebase makes isolated changes easier

- A good component balance eases finding and analyzing code
- A good component balance isolates maintenance effects
- Well-balanced translates into around 9 components of approximately equal size

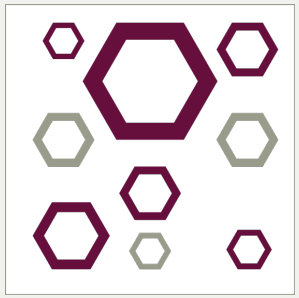
Measured by the number of components and relative size of components



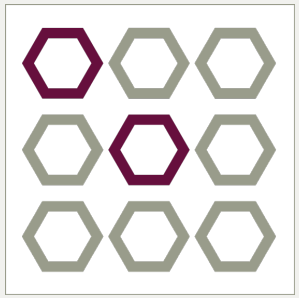
All changes in a single large component



Most changes in a single large component



Many changes scattered across multiple components



Changes isolated to one or two components of limited scope

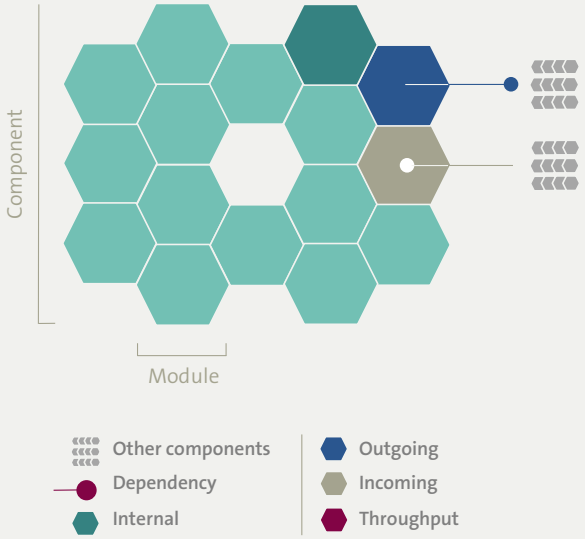
Component independence

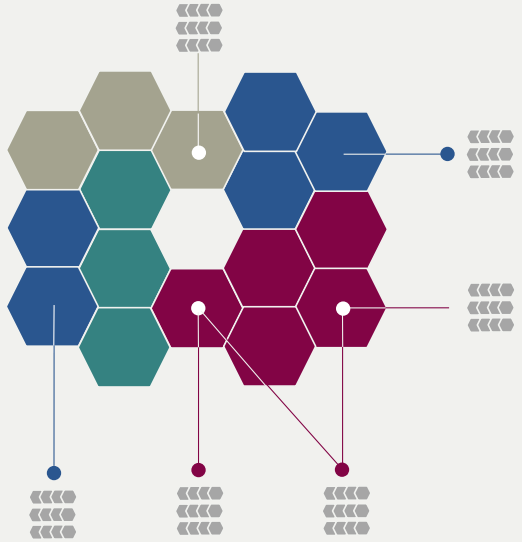
Couple architecture components loosely

Component independence captures the degree to which components can be changed and tested in isolation:

- Low component dependence allows for isolated maintenance
- Low component dependence separates maintenance responsibilities

Measured as the percentage of code which is in internal or outgoing modules within components

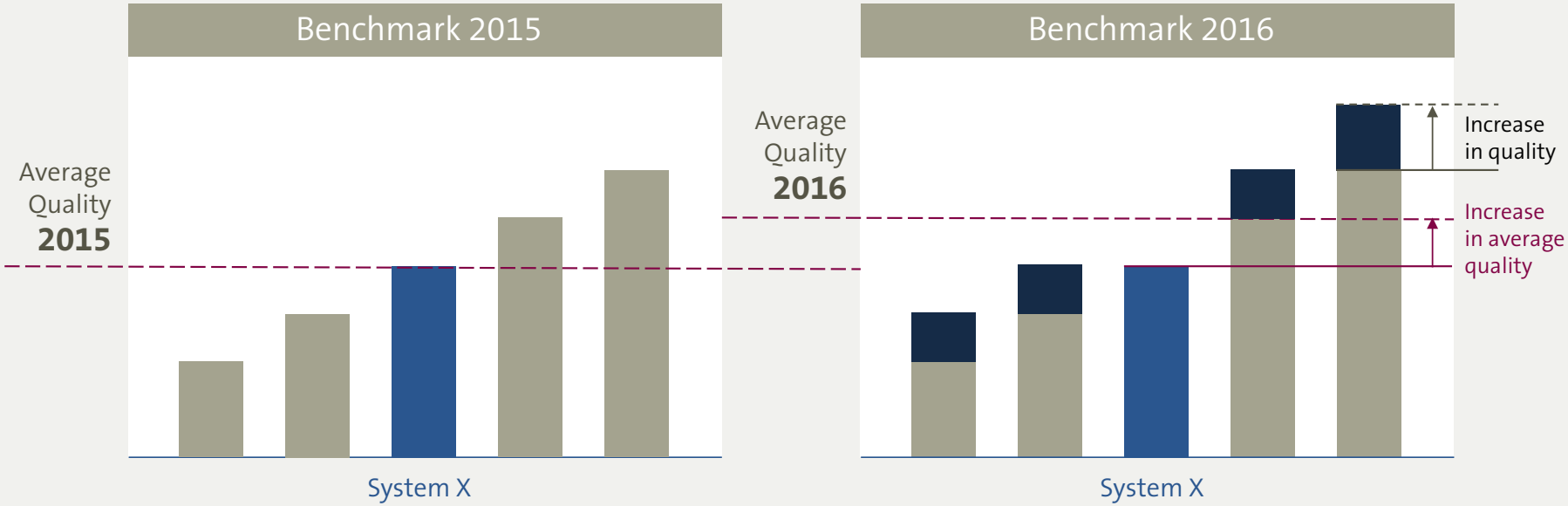




The SIG ratings represent *relative* maintainability

The SIG ratings are recalibrated yearly to reflect developments in the SIG benchmark over the past year

- The consequence of this is that the maintainability of a system will gradually go down when not maintained, as the rest of the market tends to get slightly better every year



When and how to use benchmarks

In some cases it can be useful to show where someone stands relative to a peer group (while still keeping the wider perspective in mind).

The benchmark: Everything in that meets the criteria to be considered representative for the current state of the software engineering industry.

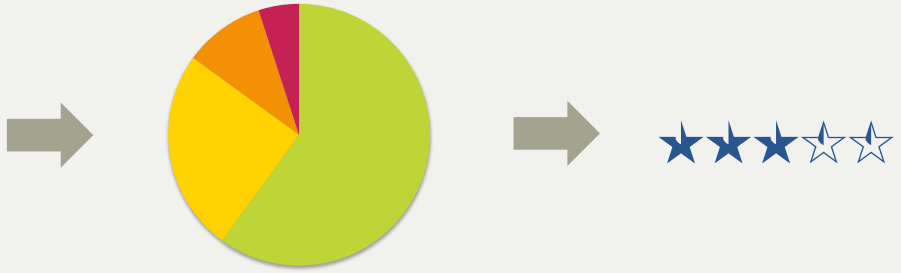
Using a risk-based approach to measure quality

Risk profiles show the percentage of code per risk category

- Unlike the number of violations, risk profiles show whether problems are incidental or structural, and also show the part of the code which is *not* at risk
- These risk profiles can then be compared against other systems from a benchmark

Example

Measurement value	Risk category
1 – 5	Low risk
6 – 10	Moderate risk
11 – 25	High risk
26+	Very high risk

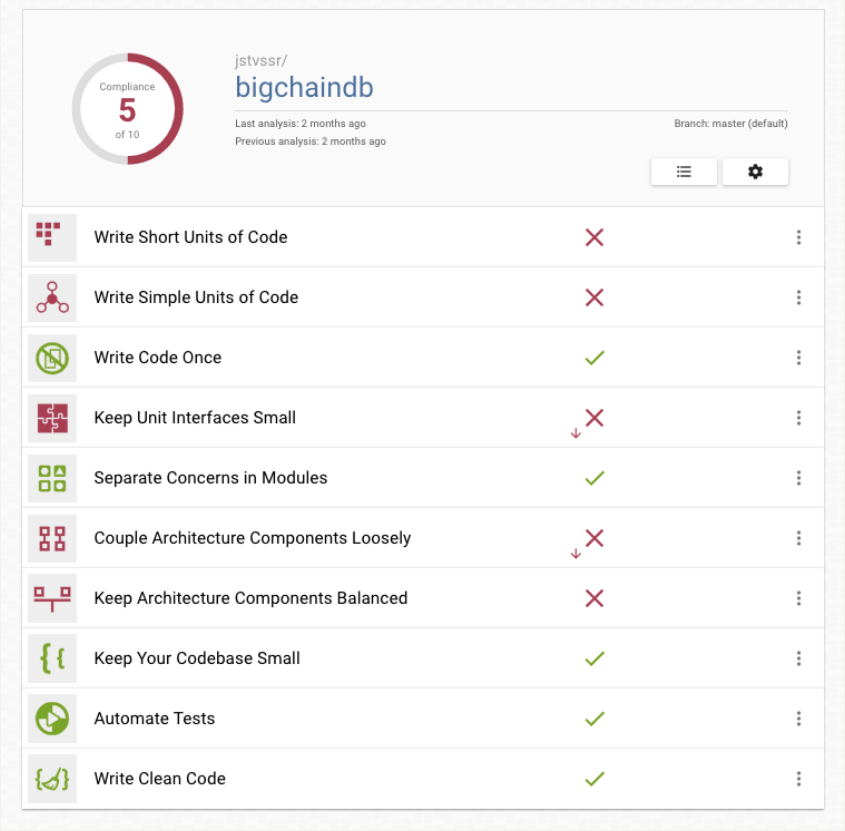


Future-proof software starts today!

"Improving maintainability does not require magic or rocket science. A combination of relatively simple skills and knowledge, plus the discipline and environment to apply them, leads to the largest improvement in maintainability."

"Maintainability is not an afterthought, but should be addressed from the very beginning of a development project. Every individual contribution counts."

Using a limited set of guidelines keeps the focus on key areas



Software Improvement Group

Contact Yiannis Kanellopoulos

+30 6938119424

y.kanellopoulos@sig.eu

@sig_eu, @ykanellopoulos